**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«Сибирский государственный университет науки и технологий
имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**КУРСОВАЯ РАБОТА**

Алгоритмы и структуры данных

Программная реализация поиска пути в лабиринте

---

отлично

Руководитель _____ 21.12.24 _____ В.В. Тынченко
подпись, дата — инициалы, фамилия

Обучающийся БПИ23-01, 23151419 _____ 21.12.24 _____ А.С. Полежаев
номер группы, зачетной книжки — подпись, дата — инициалы, фамилия

Красноярск 2024 г.

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**ЗАДАНИЕ**

на курсовую работу по дисциплине «Алгоритмы и структуры данных»

студенту Полежаеву Алексею Сергеевичу

Группа БПИ23-01 Форма обучения очная

1. Тема работы. Программная реализация поиска пути в лабиринте

2. Срок сдачи студентом работы. 10.01.25

3. Перечень вопросов, подлежащих разработке при написании теоретической части.

Рассмотреть определение лабиринта.

Разобрать методы, позволяющие найти путь в лабиринте

4. Перечень вопросов, подлежащих разработке при написании практической части (либо указать номер варианта задания)

Выбор методов нахождения пути в графе-лабиринте.

Реализация методов.

Описание структуры программы.

Описание интерфейса программы.

Тестирование программы.

5. Дата выдачи задания. 09.09.24

Руководитель. Тынченко В.В. _____

Подпись

Задание принял к исполнению (дата). 14.09.24 _____

(подпись студента)

# СОДЕРЖАНИЕ

ВВЕДЕНИЕ ..... 4
1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ ..... 5
1.1 Понятие лабиринта ..... 5
1.2 Методы поиска выхода из лабиринта ..... 5
1.2.1 Метод полного перебора ..... 5
1.2.2 Алгоритм Тремо ..... 6
1.2.3 Метод прохода по стене ..... 6
1.2.4 Метод обнаружения тупиков ..... 8
1.2.5 Метод случайного поведения мыши ..... 8
1.2.6 Алгоритм поиска в глубину ..... 8
1.2.7 Алгоритм поиска в ширину ..... 9
1.2.8 Алгоритм Дейкстры ..... 9
1.2.9 Сравнительная характеристика ..... 11
1.3 Выводы ..... 11
2 ОПИСАНИЕ ПРОГРАММЫ ..... 13
2.1 Общая характеристика программы ..... 13
2.2 Структура программы и данных ..... 13
2.3 Интерфейс программы ..... 15
2.4 Тестирование программы ..... 16
2.5 Выводы ..... 17
3 ИСПОЛЬЗОВАНИЕ ПРОГРАММЫ ..... 18
3.1 Эксплуатация программы ..... 18
3.2 Выводы ..... 24
ЗАКЛЮЧЕНИЕ ..... 25
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ ..... 26
ПРИЛОЖЕНИЕ А ..... 27
ЛИСТИНГ ПРОГРАММЫ ..... 27
ПРИЛОЖЕНИЕ В ..... 35
ОТЧЕТ АНТИПЛАГИАТА ..... 35

# ВВЕДЕНИЕ

Ориентирование на графах является важным в применении в некоторых областях деятельности человека, в том числе поиск информации в структуре данных, представленной графом. К примеру, можно найти какой-то элемент среди многих и определить причина-следственную связь, которая ведет к этому элементу.

Целью курсовой работы является рассмотрение лабиринта, как структуры данных – граф, рассмотрение некоторых методов нахождения пути между элементами графа.

Задачи курсовой работы:

1. Изучить подробно графы и лабиринты.
2. Изучить методы нахождения пути.
3. Определить оптимальный язык программирования и среду разработки для написания программы курсовой работы.
4. Спроектировать программу для нахождения пути, описать подробно структуру и устройство программы.
5. Основываясь на 4 пункте написать исходный код программы курсовой работы.
6. Описать все возможные варианты взаимодействия пользователя с программой, привести примеры.
7. Сделать выводы написать заключение и возможный потенциал развития программы и ее использования.

Для того чтобы подробно разобраться в этом и достичь цели, в курсовой работе будут рассмотрены определения лабиринтов, графов и методов нахождения пути между их элементами. Так же в ходе лабораторной работы будет спроектирована и реализована программа, которая позволяет редактировать граф и находить пошаговый путь между его элементами.

После написания программы, она будет протестирована по принципу черного ящика.

В тексте курсовой работы будут представлены все возможные варианты того, как пользователь сможет взаимодействовать с программой.

# 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1 Понятие лабиринта

Лабиринт – это структура, состоящая из запутанных путей к выходу или путей, ведущих в тупик.

Лабиринт – это, с математической точки зрения, граф как показано на рисунке 1.1, фигура, которая состоит из точек и линий, соединяющих эти точки. Точки называют вершинами графа, а линии – его рёбрами. [3]

В настоящее время лабиринт – это головоломка, использующаяся для развития мышления, внимания и памяти.

Графы используются во многих областях, например, в навигации, хранение информации о каких-либо элементах и связей между ними.

Решение лабиринта – это поиск определенной траектории, которая соединяет вход в лабиринт и отличный от него выход. [4]

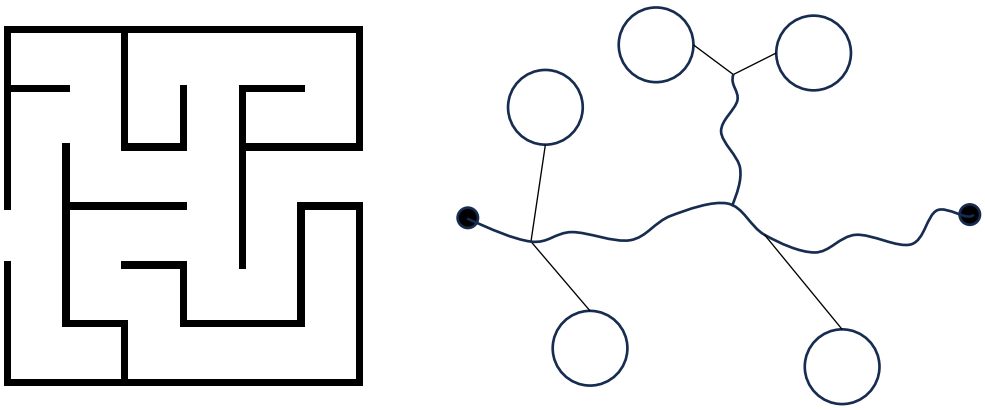


Рисунок 1.1 – Представление лабиринта в виде графа

## 1.2 Методы поиска выхода из лабиринта

– метод полного перебора (метод грубой силы);
– алгоритм Тремо;
– метод прохода по стене (метод одной стороны);
– метод обнаружения тупиков;
– алгоритм случайного поведения мыши;
– алгоритм поиска в глубину;
– алгоритм поиска в ширину;
– алгоритм Дейкстры.

### 1.2.1 Метод полного перебора

Работа алгоритма этого метода подразумевает проверку всех вариантов путей и постоянно запускается заново в том случае, если выбранный путь окажется неудачным. Поскольку этот алгоритм проводит лишние вычисления

он имеет низкую скорость работы, что делает этот метод неэффективным. В конечном итоге этот алгоритм имеет временную сложность O(n!). Несмотря на эти недостатки алгоритма, он прост в реализации, так как зачастую состоит из небольшого количества операторов, повторяющихся в цикле. [1]

Основная суть алгоритма.

1. Выбирается какой-то путь.
2. Если путь не оказался верным, алгоритм запускается заново только выбирается ещё не пройденный путь.
3. Если путь оказался верным алгоритм завершает свою работу.

### 1.2.2 Алгоритм Тремо

Этот алгоритм был придуман Шарлем Пьером Тремо (Charles Pierre Trémaux). Этот метод реализует поиск выхода из лабиринта, за счёт того, что он обозначает путь, рисует линии и точки по всему лабиринту. Этот алгоритм работает при соблюдении следующих правил. [1]

1. Выбирается случайный проход и алгоритм следует по нему до следующей развилки.
2. Алгоритм помечает начало и конец каждого прохода по мере его прохождения.
3. Если алгоритм идет по проходу первый раз он помечает его одной точкой.
4. Если алгоритм зашел в тупик, он возвращается по тому же пути до первой развилки, после чего пометив его второй раз.
5. Проход, который отмечен дважды, больше не подлежит проходу, теперь он считается тупиком.
6. На каждой развилке выбирается проход, отмеченный наименьшим количеством раз, лучше вообще не отмеченный.

Этот алгоритм эффективнее, чем алгоритмы полного перебора и метода случайного поведения мыши. Однако для достижения этой эффективности он имеет более сложную реализацию, так как алгоритму необходимо помечать каждую пройденную развилку.

### 1.2.3 Метод прохода по стене

Следовать вдоль стены или поворачивать всегда только в одну сторону – один из наиболее широко известных методов поиска выхода из лабиринта. Благодаря условию внешней связи лабиринта, которая состоит в том, что все внутренние стены головоломки соединены с внешней стеной. Если это условие выполняется, то следуя всегда только в одну сторону всегда можно выйти из лабиринта. Если же условие не выполнимо, это не значит, что алгоритм со 100% вероятностью не будет находить выход, все же в некоторых случаях удастся найти верный путь. Можно сказать, что, если условие выполняется, алгоритм полезен при поиске решения головоломки. Также можно сказать, что метод эффективен, так как он всего лишь следует по какой-то одной стороне, а для этого не требуется помечать каждую развилку, еще при попадании в тупик не требуется заново запускать алгоритм. [2]

Основная суть алгоритма.

1. Двигаемся вперед до следующей развилки или тупика.

2. На развилке всегда поворачиваем в одну сторону, если все же зашли в тупик, то возвращаемся до развилки и снова поворачиваем в том же направлении.

3. Завершаем алгоритм при выходе из лабиринта.

Реализация этого метода может быть относительно нетрудной. В конечном итоге на выполнение такого алгоритма понадобится на много меньше времени чем у метода грубой силы и метода мыши. Однако тот факт, что этот алгоритм нельзя применить ко всем лабиринтам, делает его необходимым только в определённых ситуациях. Блок-схема реализации этого алгоритма представлена на рисунке 1.2.

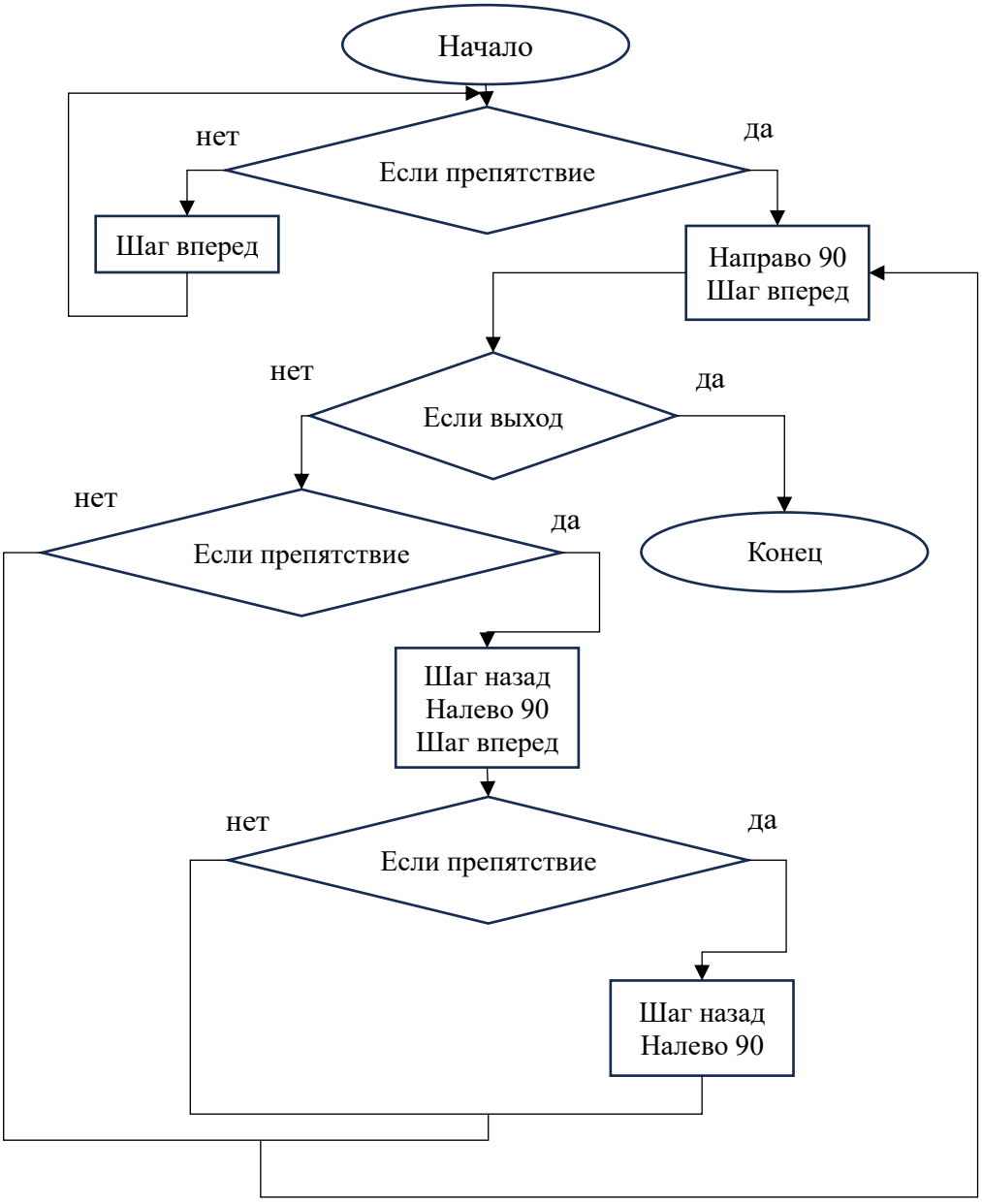



Рисунок 1.2 – Алгоритм прохода по стене

### 1.2.4 Метод обнаружения тупиков

Метод обнаружения тупиков, чем-то похож на метод полного перебора. Это логично, ведь если алгоритм перебора перебирает варианты пока не найдет верный, а обнаружение тупиков проверяет все варианты. Алгоритм ищет все тупики и помечает их, оставляя только пути, являющиеся решением головоломки. [1]

Основная суть алгоритма.

1. Находим тупик. Заполняем путь в обратном направлении от тупика до развилки.
2. Ищем следующий тупик и снова заполняем до развилки.
3. Алгоритм повторяется до того момента, пока не будут заполнены все тупики. Оставшиеся пути будут являться верными.

Время выполнения алгоритма сравнимо с зависимостью $O(n^2)$. Хоть алгоритм и выполняется очень долго, он может найти все пути к выходу из лабиринта, если их несколько. К тому же он не требует дополнительной памяти.

### 1.2.5 Метод случайного поведения мыши

Этот алгоритм полагается на волю случая. Из названия можно понять, что алгоритм в действительности похож на поведение мыши, которая попала в лабиринт и наугад пытается найти из него выход. Из-за того, что на каждой развилке алгоритм случайным образом выбирает направление движения, время выполнения становится очень большим или наоборот очень быстрым, потому что может повезти и алгоритм с первой попытки найдет выход, в таком случае временная зависимость будет O(1). [1]

Основная суть алгоритма.

1. Следуем вперед до развилки.
2. Случайным образом выбираем направление и следуем ему.
3. Повторяем процедуру на каждой развилке пока не выйдем из лабиринта.
4. Завершаем алгоритм при выходе из лабиринта.

### 1.2.6 Алгоритм поиска в глубину

Данный алгоритм является рекурсивным и работает по следующему принципу:

1. Пройти по первому попавшемуся ребру из первоначальной вершины в вершину N.
2. Если N не совпадает с конечной вершиной и у нее нет пометки, что она просмотрена, то она становится начальной и алгоритм вызывает сам себя из этой точки.
3. Если не обнаружилось смежных не помеченных вершин, алгоритм возвращается к вершине, из которой он спустился в эту и проверяет другие смежные. [5]

### 1.2.7 Алгоритм поиска в ширину

Метод используется для нахождения кратчайшего пути. Алгоритм также как и алгоритм поиска в глубину является рекурсивным. Работает он по следующему принципу:

1. Из первоначальной вершины проходим по всем смежным вершинам.
2. Если среди них нет нужной финишной вершины, то поиск осуществляется проходом по смежным вершинам к этим, в той последовательности в которой они рассматривались.

Основная суть алгоритма в том, что вершина, которая посещена раньше, будет использована раньше. [6]

### 1.2.8 Алгоритм Дейкстры

Так как можно считать, что лабиринт – это граф, мы можем использовать алгоритм Дейкстры. Суть алгоритма заключается в том, что благодаря ему мы можем найти кратчайший путь от начальной вершины графа до любой другой. Алгоритм не перебирает все варианты, а действует пошагово. На каждом следующем шаге он выбирает вершину, которая наименее отдалена от текущей. Таким образом он проходит пока не достигнет цели. На рисунке 1.3 представлена блок-схема этого алгоритма. [2]

Основная суть алгоритма.

1. Инициализируется два множества. Множество для обработанных вершин. Множество, которое содержит все остальные вершины.
2. Каждой вершине присваивается вес, он показывает минимальную сумму путей от исходной точки до текущей. Для исходной вершины это 0. Для вершин с еще неизвестным значением веса, присваивается значение бесконечности.
3. На каждом шаге алгоритм выбирает вершину из множества с наименьшим весом, помещает эту вершину в множество обработанных и прописывает веса для соседних вершин относительно текущей.
4. Алгоритм продолжает действовать, пока не посетит все вершины или пока не дойдет до конца.

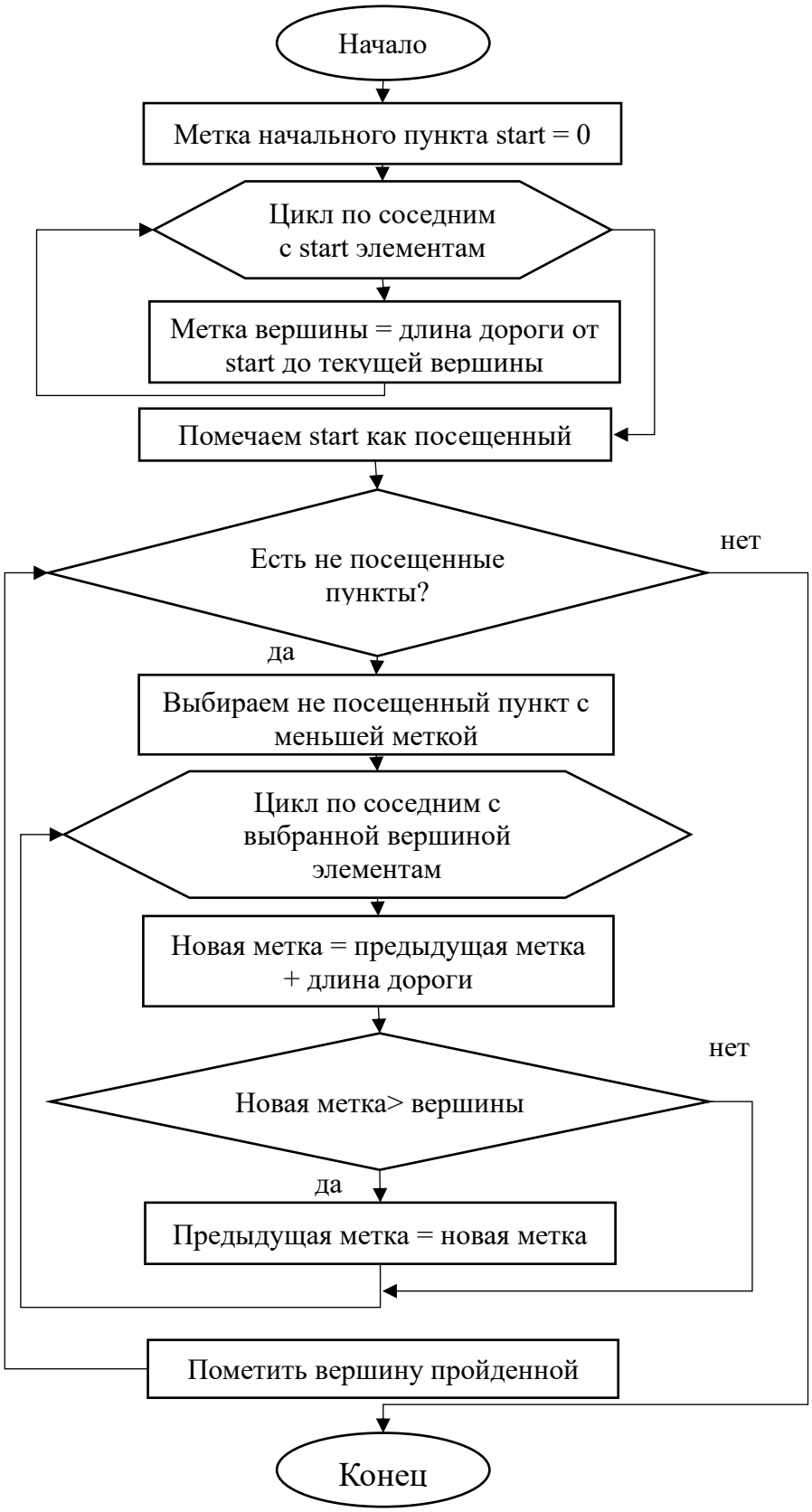



Рисунок 1.3 – Алгоритм Дейкстры

### 1.2.9 Сравнительная характеристика

Перечисленные алгоритмы без сомнения находят выход из лабиринта. Однако, каждый делает это по-своему и имеет свои плюсы и минусы. В таблице 1 приводятся основные плюсы и минусы рассматриваемых методов.

Таблица 1 – Плюсы и минусы

| Алгоритм | Плюсы | Минусы |
|---|---|---|
| Полный перебор | Прост в реализации. | Очень большое время выполнения. |
| Тремо | Имеет самую высокую скорость, среди перечисленных методов. | Имеет сложности в реализации. |
| Проход по стене | Прост в реализации. | Подходит не для всех видов лабиринтов. Скорость не самая высокая. |
| Обнаружение тупиков | Может найти несколько решений. | Имеет сложности в реализации. Время выполнения очень большое. |
| Поведение мыши | Прост в реализации. | Скорость выполнения зависит от воли случая. |
| Поиск в глубину | Прост в реализации | Довольно быстрый. |
| Поиск в ширину | Прост в реализации, находит кратчайший путь | Имеет примерно такую же скорость, что и поиск в глубину |
| Алгоритм Дейкстры | Находит кратчайший из путей, если их несколько. | Могут возникнуть сложности с пониманием и реализацией алгоритма. Используется со взвешенными графами. |

## 1.3 Выводы

В данном разделе были изучены некоторые методы, позволяющие найти выход из лабиринта. Рассмотрев все характеристики этих методов, можно сделать вывод, что каждый из них имеет как хорошие стороны, так и плохие. Метод полного перебора прост и надежен, но может оказаться очень затратным по времени и ресурсам при больших размерах лабиринта. Алгоритм Тремо будет значительно быстрее и позволяет сэкономить на лишних повторных проходах. Метод следования по стене надежен, однако он не подходит для всех типов лабиринтов. Обнаружение тупиков позволяет найти несколько путей, если они существуют, но из-за того, что он перебирает все варианты, он проигрывает по времени. Метод случайного поведения мыши хоть и находит выход, но не известно, когда это вообще произойдет. Методы поиска в глубину и в ширину сочетают в себе простоту в реализации и хорошие скоростные характеристики. Алгоритм Дейкстры является одним из самых эффективных методов, он находит кратчайший путь в графе и может быть применен для решения лабиринта.

Для реализации в программе курсовой работы были выбраны алгоритмы поиска в глубину и в ширину. Выбор обусловлен тем, что методы подходят для неориентированных невзвешенных графов, они просты в реализации и имеют разумную скорость выполнения.

# 2 ОПИСАНИЕ ПРОГРАММЫ

## 2.1 Общая характеристика программы

Выбор языка программирования для написания программы, пал на язык C++. Программа, которая была создана входе курсовой работы, предназначена для исследования различных методов прохода по графам. Программа представляет собой консольное приложение. Во время написания программы ей было присвоено название *Graph_Labirint.* Функционал программы соответствует названию данной курсовой работы. На вход программа получает лабиринт-граф, представленный в виде списков смежности, а на выходе выдает пошагово путь из начала в конец.

Программа была сделана с использованием такого инструмента как Visual Studio 2022 версии 17.7.3. Программа не требовательная и не требует высоких конфигураций компьютера. Они представлены ниже.

– операционная система. Windows 7, 8, 10, 11 (64-разрядная версия);

– процессор с двумя ядрами и тактовой частотой не ниже 1,8 ГГц;

– оперативная память не менее 4 ГБ;

– место на диске не менее 512 МБ.

## 2.2 Структура программы и данных

Рассмотрим структуру программы курсовой работы. Для достижения решения поставленной задачи данной курсовой работы был выбран для реализации объектно-ориентированный подход. Программа будет работать в двух режимах. В первом режиме будет выполняться проход, выбранным пользователем методом, по графу-лабиринту, заданному из файла, на выход программа выдаст время выполнения прохода и пошаговый путь от вершины-старта до вершины-финиша. Во втором режиме пользователь сам задаёт граф-лабиринт, старт и финиш, а также выбирает каким методом прохода воспользоваться. На выходе во втором режиме также получается время выполнения и пошаговый путь.

Вся программа состоит из трех файлов с кодом. *Graph.h, Graph.cpp, main.cpp*. В файле Graph.h содержится объявление класса Graph и его методов, для представления графа-лабиринта в виде списков смежности. В файле Graph.cpp содержится реализация каждого метода класса Graph. В файле main.cpp содержится реализация пользовательского интерфейса и использования методов класса Graph, также в нем реализована функция вывода последовательности пути в наглядном для человека виде. Для наглядного представления структуры данных класса Graph была создана диаграмма классов, которая представлена на рисунке 2.1.

Для реализации программы курсовой работы, в ней были реализованы методы представления и обработки такой структуры данных как граф. Это методы. конструктор по умолчанию, конструктор с параметром и конструктор копий, метод ввода in() и метод вывода графа в виде списка смежности out() оба этих метода могут работать как с консолью, так и с файлом, также для них

были перегружены соответствующие операторы >>, <<, метод добавления вершины графа addVertex() и метод удаления вершины delVertex(), методы добавления и удаления ребра графа соответственно addEdge() и delEdge(), методы, которые находят путь из вершины a в b – DFS(), BFS() и метод вывода пути на экран – wayOut(). Назначение этих методов класса Graph и их входные и выходные данные описаны в таблице 2.1 – Описание методов класса Graph. Полный код программы курсовой можно увидеть в приложении А.

Рисунок 2.1 – Диаграмма классов

В классе граф содержатся некоторые поля, необходимые как для представления графа в виде списков смежности, так и для работы некоторых методов.

Таблица 2.1 - Описание методов класса Graph

| **Название метода** | **Назначение** | **Тип входных данных** | **Тип возвращаемых данных** |
|---|---|---|---|
| in | Ввод данных для элемента типа Graph, как с консоли, так и с файла. | istream& | void |
| out | Вывод элемента Graph в виде списков смежности, как в консоль, так и в файл. | ostream& | void |
| addVertex | Добавление новой вершины и добавление ей связи с другими вершинами. | - | void |

Окончание таблицы 2.1

| Название метода | Назначение | Тип входных данных | Тип возвращаемых данных |
|---|---|---|---|
| delVertex | Удаление всех ребер, выходящих из вершины с указанным индексом. Удаление самой вершины. | int | void |
| addEdge | Добавление ребра между двумя вершинами с указанными индексами. | int, int | void |
| delEdge | Удаление ребра между двумя вершинами с указанными индексами. | int, int | void |
| DFS | Поиск пути из вершины-старта в вершину-финиш (заполнение вектора way). | int, int | void |
| BFS | Поиск пути из вершины-старта в вершину-финиш (заполнение вектора way). | int, int | void |
| wayOut | Вывод пути на экран | - | void |

### 2.3 Интерфейс программы

Вся программа курсовой работы реализована в виде консольного приложения. При запуске этого приложения пользователю будет предоставлен выбор в каком из двух режимов продолжить работу.

Режим первый предоставляет меню с выбором из трех пунктов. Каждый пункт соответствует методу нахождения пути в графе-лабиринте. После выбора какого-либо пункта программа выполнит проход от точки а к точке б, выбранным пользователем методом и выведет время выполнения этой операции и путь, который она нашла.

Режим второй предоставляет пользователю с помощью некоторых манипуляций задать собственный граф-лабиринт. После чего пользователю также как и в первом режиме предоставится меню из трех пунктов, для выбора метода нахождения пути и также после нахождения пути выведет время и сам путь.

Третьим пунктом среди выбора режимов пользователю предоставится возможность завершить выполнение программы.

Для наглядного представления меню, на рисунке 2.2 представлена UML – диаграмма.


Рисунок 2.2 – UML – диаграмма предоставленных вариантов пользователю

## 2.4 Тестирование программы

Программа курсовой работы была протестирована методом «чёрного ящика». Результаты тестирования предоставлены в таблице 2.2.

Тестирование методом «чёрного ящика» – это проверка, при которой тестировщик не имеет доступа к коду. Он, как реальный клиент или пользователь, оценивает функции и работу программы, ориентируясь исключительно на интерфейс взаимодействия.

Таблица 2.2 – Тесты программы

| № | Тип теста | Вывод | Результат |
|---|---|---|---|
| 1 | Выбор первого режима | Меню первого режима | Положительный |
| 2 | Выбор второго режима | Меню второго режима | Положительный |
| 3 | Выбор третьего режима | Завершение программы | Положительный |
| 4 | Выбор режима больше 3 | Завершение программы | Положительный |
| 5 | Ввод символа, вместо номера режима | Повторное приглашение к вводу режима | Положительный |
| 6 | Ввод списка смежности (добавление вершины с индексом, превосходящим диапазон) | Ошибка | Отрицательный |
| 7 | Выбор метода нахождения пути | Время, путь | Положительный |
| 8 | Добавить вершину | Вершина добавлена | Положительный |

Окончание таблицы 2.2

| № | Тип теста | Вывод | Результат |
|---|---|---|---|
| 9 | Добавить вершину и ввести символ | Ошибка типов данных | Отрицательный |
| 10 | Удалить вершину с существующим индексом | Вершина удалена | Положительный |
| 11 | Удалить вершину с несуществующим индексом | Ошибка | Отрицательный |
| 12 | Завершить программу в каком-либо режиме | Выбор режима или проскакивание цикла | Отрицательный |

### 2.5 Выводы

В этом разделе курсовой работы было приведено описание структуры программы, её требования к конфигурациям компьютера, описаны все основные методы, использующиеся в работе программы, показана диаграмма задуманного взаимодействия для пользователя, были проведены некоторые тесты.

Из проведенных тестов следует сделать вывод, что если вводить данные в программу правильно, то она будет работать корректно.

# 3 ИСПОЛЬЗОВАНИЕ ПРОГРАММЫ

## 3.1 Эксплуатация программы

В разделе рассматриваются варианты взаимодействия пользователя с программой курсовой работы. Далее представлен весь функционал данной программы с наглядными примерами и результатами выполнения какого-либо пункта программы.

Запустив программу, пользователь увидит главное меню, оно представлено на рисунке 3.1. Для продолжения работы, пользователь должен выбрать один из предложенных режимов в меню, сделать он это может, нажав на кнопку с цифрой на клавиатуре, соответствующей режиму:

– заполнить граф из файла и выбрать способ нахождения пути;

– режим - сформировать граф из данных пользователя и выбрать способ нахождения пути;

– любая другая клавиша для завершения работы программы.

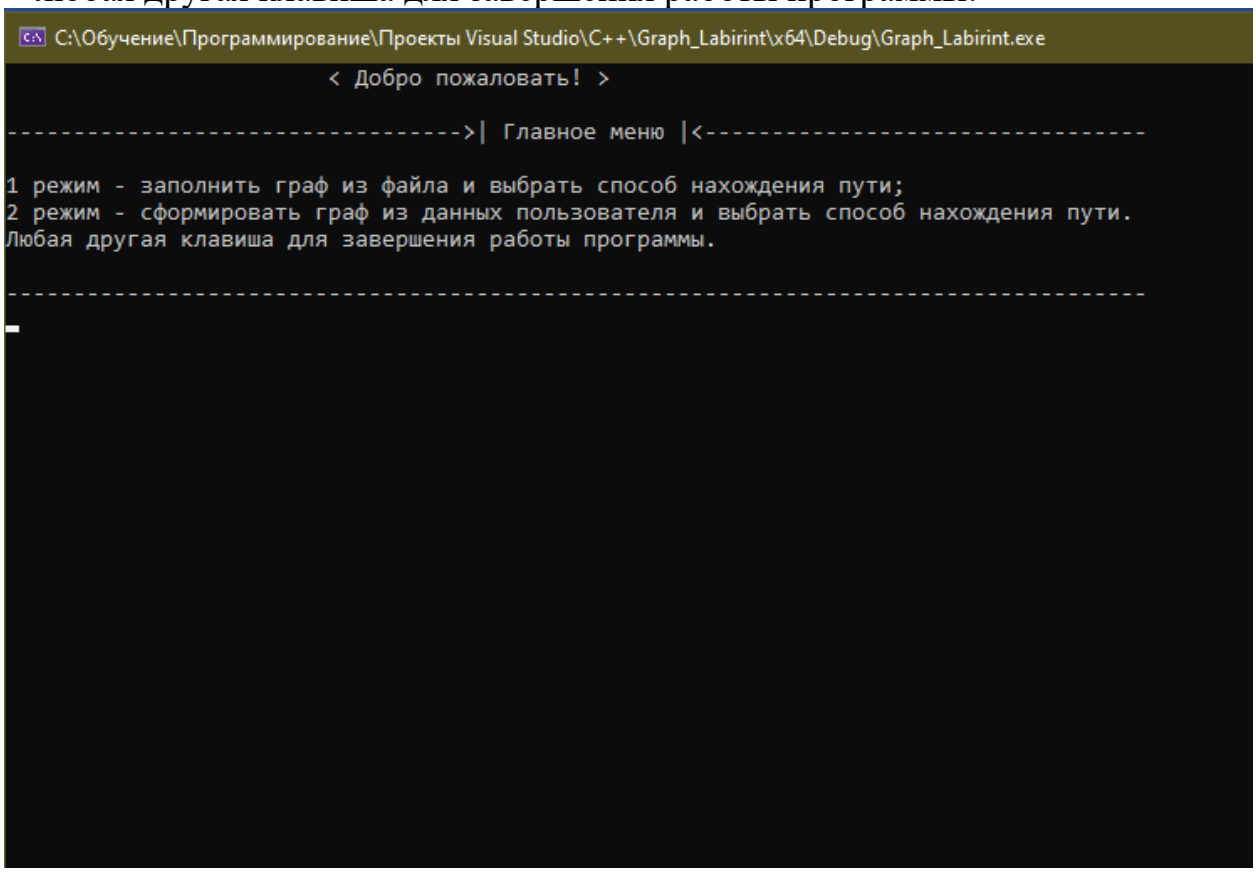



Рисунок 3.1 – Главное меню программы

Наглядный пример графа, заданного из файла, представлен на рисунке 3.2, а лабиринт, представлением которого является этот граф, представлен на рисунке 3.3.

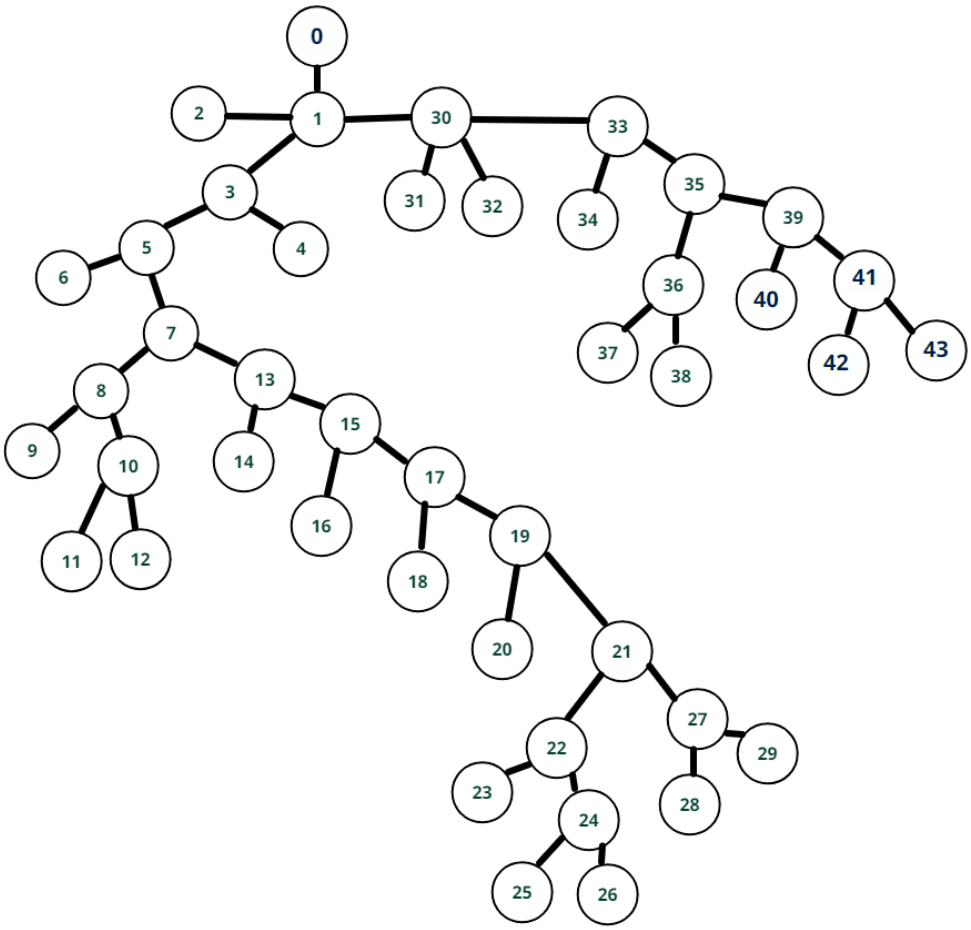



Рисунок 3.2 – Лабиринт в виде графа

Для представления лабиринта в виде графа считалось, что в лабиринте начало, конец, каждый тупик и каждая развилка является вершинами графа. После определения всех вершин графа, они соединяются в том же порядке, как и в изначальном лабиринте. Таким образом получаем лабиринт, записанный в виде неориентированного невзвешенного графа.

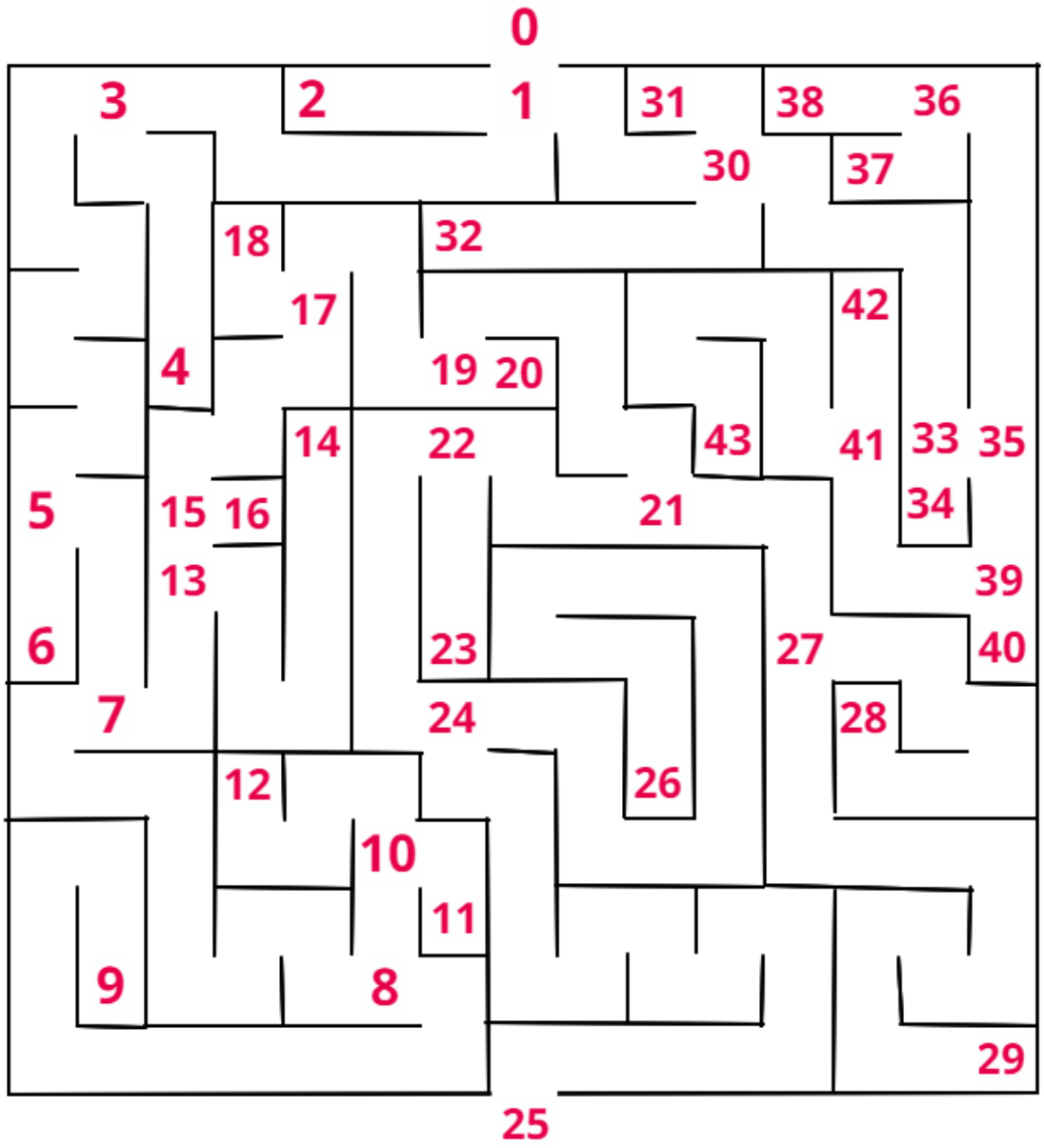



Рисунок 3.3 – Лабиринт, из которого получился граф

При выборе первого режима работы программы в главном меню, программа переходит в меню работы с графом, заданным из файла, пример работы программы представлен на рисунке 3.4. Пользователю показывается граф, представленный в виде списков смежности, и пользователю предлагается на выбор два алгоритма поиска пути между вершинами.

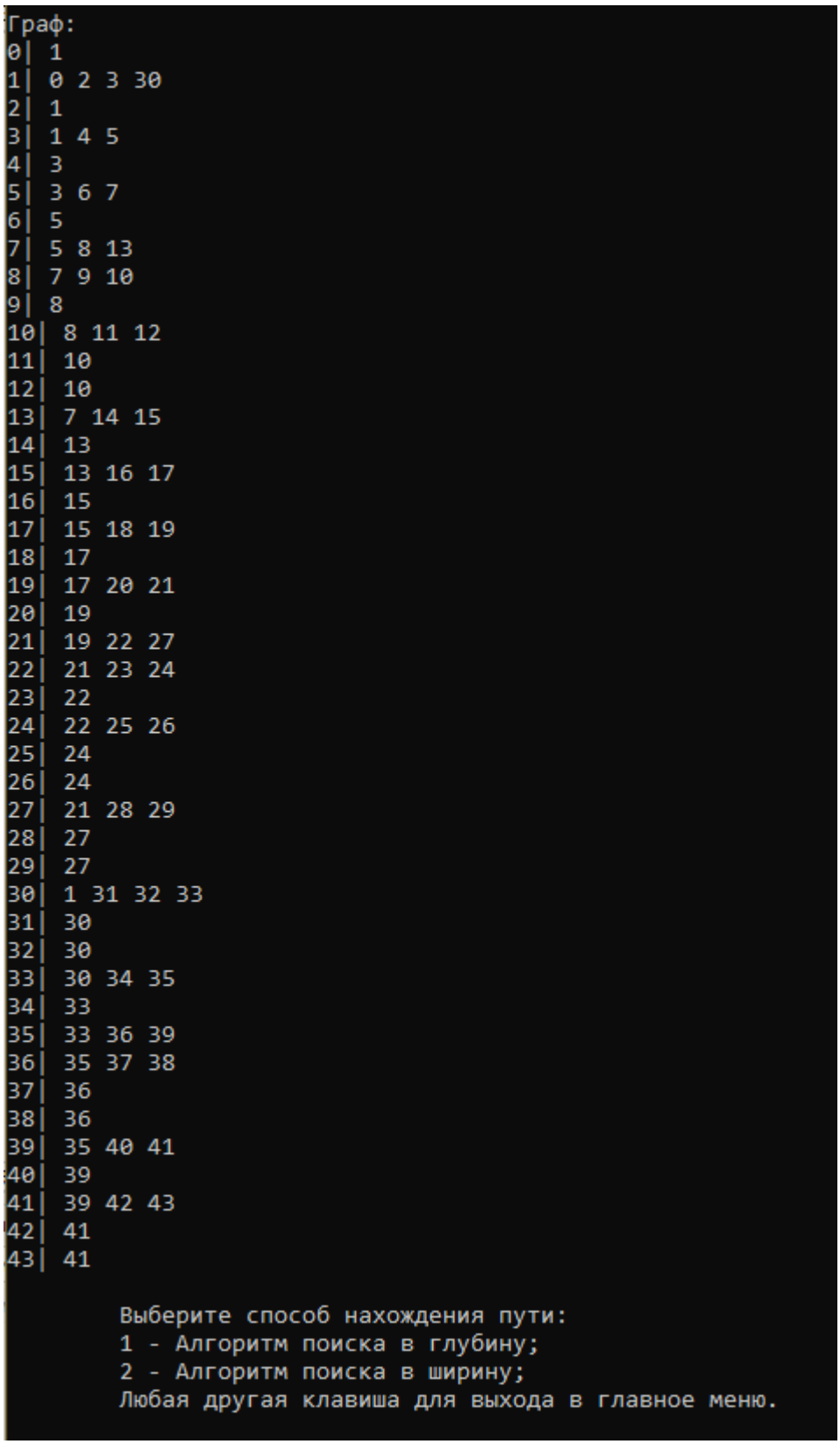



Рисунок 3.4 – Меню работы с графом из файла

После выбора алгоритма поиска, пользователю предлагают ввести стартовую и конечную вершины, после чего выводится путь между ними, найденный выбранным алгоритмом. Это показано на рисунках 3.5 и 3.6.

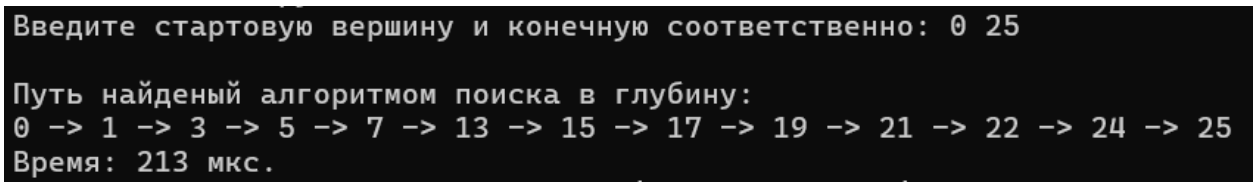



Рисунок 3.5 – Поиск пути в глубину между заданными вершинами графа из файла

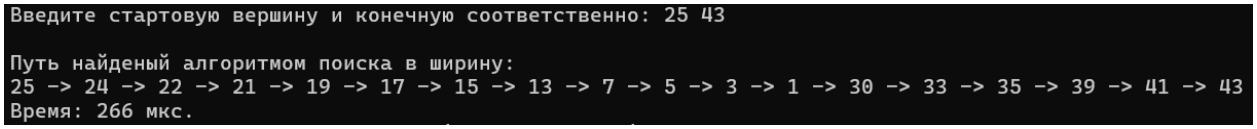



Рисунок 3.6 – Поиск пути в ширину между заданными вершинами графа из файла

После выполнения работы первого режима программа снова переходит в главное меню.

При выборе второго режима работы программы в главном меню, программа переходит в меню работы с графом, заданным пользователем с клавиатуры. Пользователю предлагается ввести граф с клавиатуры это показано на рисунке 3.7, после чего предлагаются различные способы отредактировать этот граф и найти в нем путь, что показано на рисунке 3.8. На рисунке 3.9 представлены примеры редактирования графа.

Наглядное представление вводимого графа и добавленной к нему вершины, в этом режиме, предоставлено на рисунке 3.10.

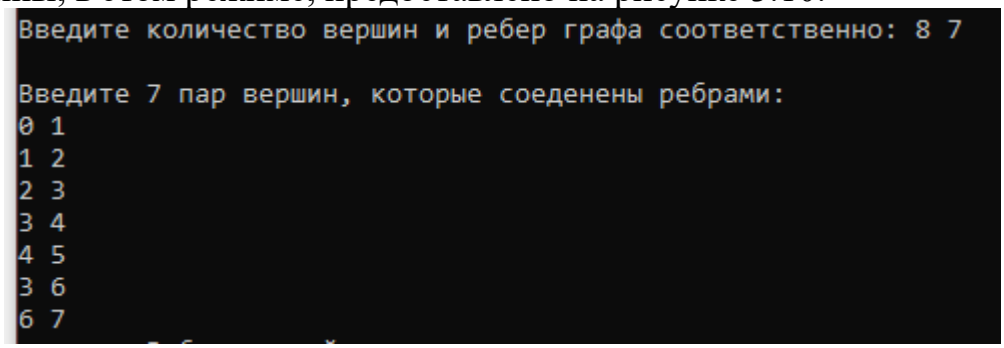



Рисунок 3.7 – Ввод графа с клавиатуры

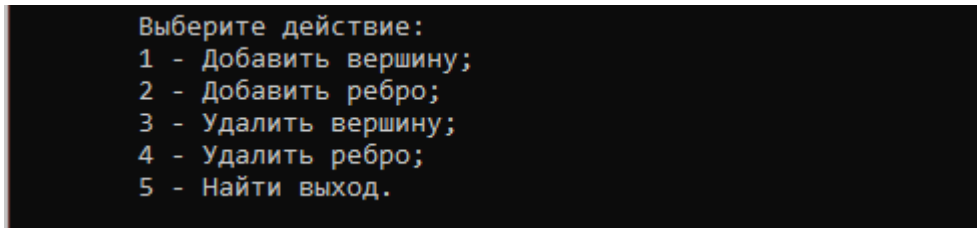



Рисунок 3.8 – Редактирование графа

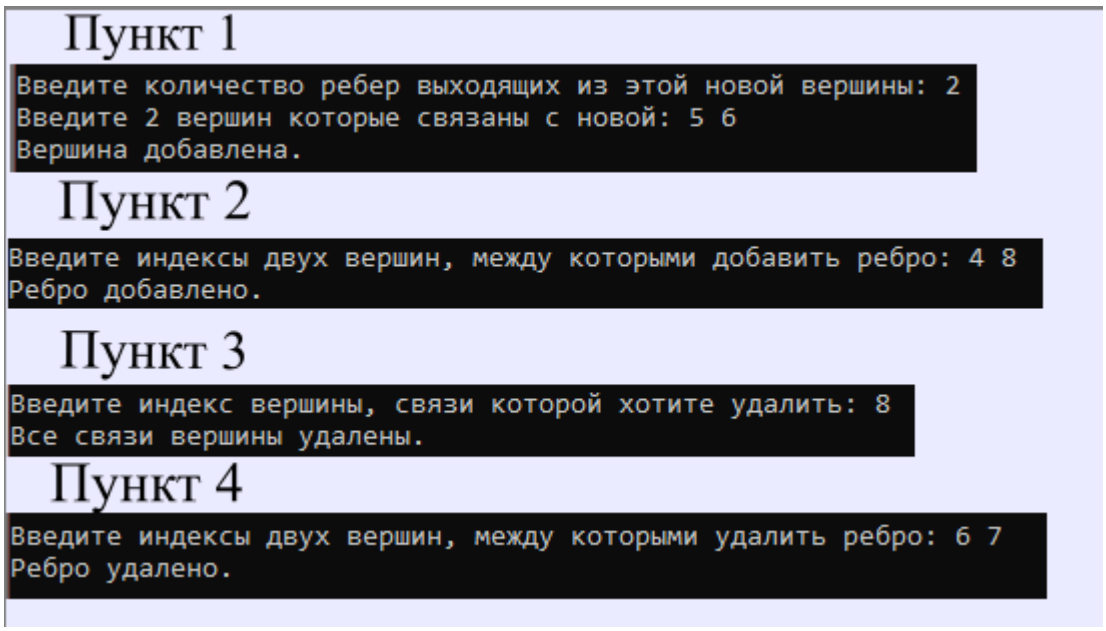



Рисунок 3.9 – Варианты редактирования введенного графа

При выборе пятого пункта пользователю показывается граф, представленный в виде списков смежности и пользователю предлагается на выбор два алгоритма поиска пути между вершинами. Результаты работы программы представлены на рисунках 3.11, 3.12 и 3.13.

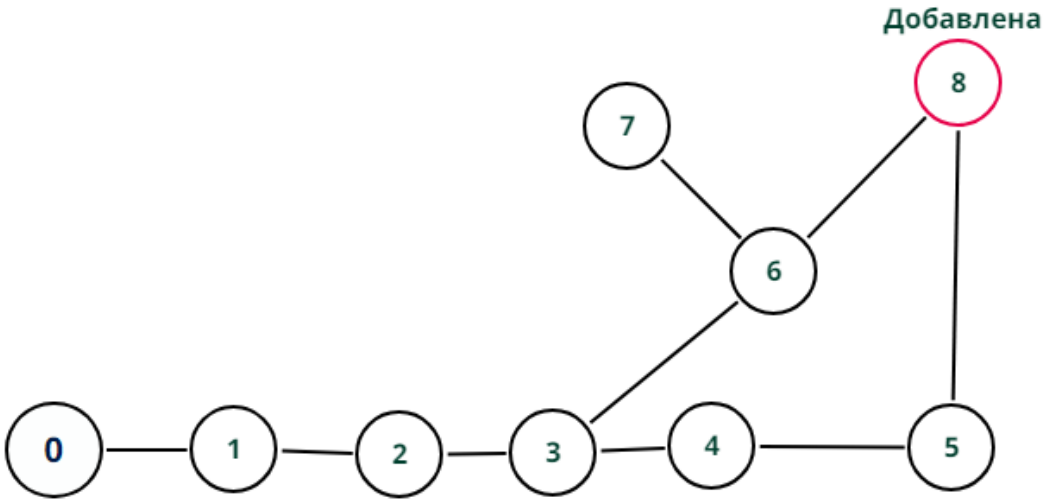



Рисунок 3.10 – Граф введенный с клавиатуры

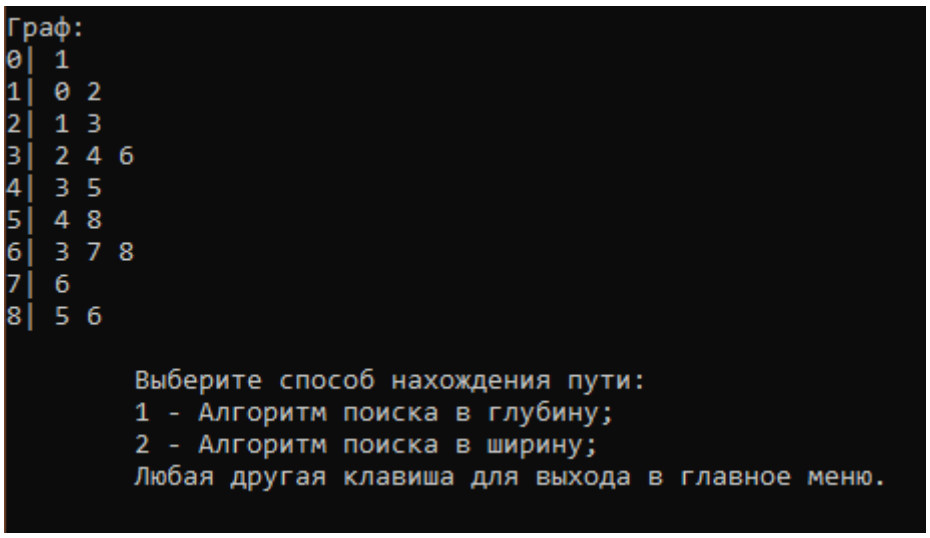



Рисунок 3.11 - Граф введенный с клавиатуры

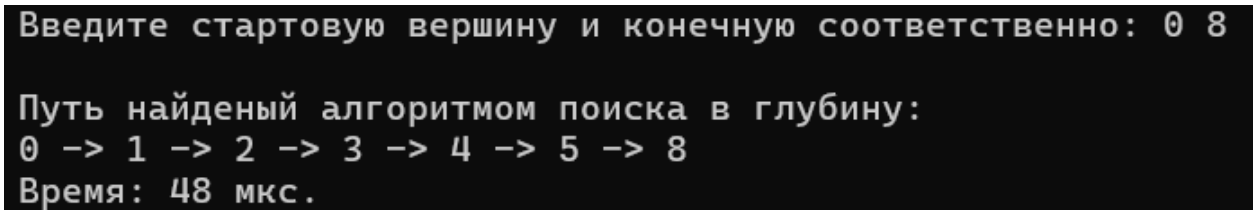



Рисунок 3.12 - Поиск пути в глубину между заданными вершинами графа пользователя

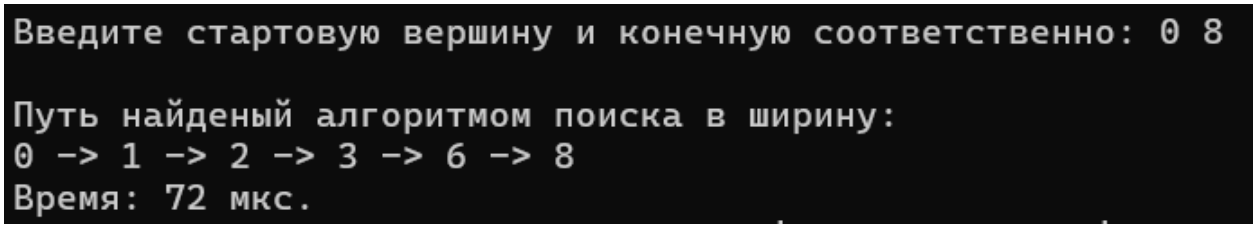



Рисунок 3.13 – Поиск пути в ширину между заданными вершинами графа пользователя

После выполнения работы второго режима программа снова переходит в главное меню.

### 3.2 Выводы

В данном разделе курсовой работы были рассмотрены все возможные варианты использования программы пользователем. Так же было наглядно продемонстрировано представление данных для ввода в программу, а именно, представление лабиринта в виде графа и ввод графа в программу в виде списков смежности.

# ЗАКЛЮЧЕНИЕ

Подводя итог курсовой работы, можно сказать, что в работе были рассмотрены основные методы нахождения пути в графе-лабиринте и некоторые из них были реализованы в программе курсовой работы.

Для написания курсовой работы бы выполнены следующие пункты:

– дано определение лабиринта;

– рассмотрены основные методы нахождения пути;

– спроектирована программа и описана ее структура с использованием UML – диаграмм и диаграммы класса;

– написана программа на языке C++, в которой реализованы некоторые методы нахождения пути;

– проведены некоторые тесты по принципу черного ящика;

– представлены все возможные варианты использования программы пользователем.

Программа, написанная в ходе курсовой работы, способна находить одним из методов кратчайший путь между вершинами графа. Я считаю, что если усовершенствовать программу, например, добавить возможность определять сплетения дорог со снимков спутника и представлять их в виде графа, то можно использовать ее в качестве нахождения кратчайшего пути от одного пункта до другого пункта на карте. Также в дополнение к этому, можно представлять сплетения дорог в виде взвешенного графа, где ценой ребра будет уровень пробки на дороге, и добавить метод нахождения пути алгоритмом Дейкстры, тогда программа будет способна находить кратчайший путь с учетом пробок на дорогах. Такие возможности могут пригодиться в онлайн навигаторе.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Хабр: «Лабиринты: классификация, генерирование, поиск решений»: сайт – 2006. – URL: https://habr.com/ru/articles/445378/ (дата обращения: 20.09.2024). – Текст: электронный
2. CYBERLENINKA: «Реализация алгоритма поиска выхода из лабиринта»: сайт – 2021. – URL: https://cyberleninka.ru/article/n/realizatsiya-algoritma-poiska-vyhoda-iz-labirinta/viewer (дата обращения: 03.10.2024). – Текст: электронный
3. Конфорович А.Г.: «Математика лабиринта», 1987. – Текст: непосредственный
4. НОУ ИНТУИТ: «Графы и их применение. Лекция 4: Эйлеровы графы»: - сайт. – 2003. – URL: https://intuit.ru/studies/courses/58/58/lecture/1714?page=3 (дата обращения: 03.10.2024). – Текст: электронный
5. Хабр: «Реализуем алгоритм поиска в глубину»: - сайт. – 2006. – URL: https://habr.com/ru/companies/otus/articles/660725/ (дата обращения: 16.10.2024). – Текст: электронный
6. Хабр: «Базовые алгоритмы на графах»: - сайт. – 2006. – URL: https://habr.com/ru/companies/timeweb/articles/751762/ (дата обращения: 16.10.2024). – Текст: электронный

# ПРИЛОЖЕНИЕ А

# ЛИСТИНГ ПРОГРАММЫ

Содержание файла Graph.h

```
#pragma once
#include<iostream>
#include<algorithm>
#include<list>
#include<vector>
#include<queue>

using namespace std;

#ifndef GRAPH_H
#define GRAPH_H

// граф представляется в виде списков смежности
class Graph {
	int vertexCount, edgeCount;
	vector<vector<int>> matrix;
	vector<int> way;
	vector<int> visited;

public:
	Graph();
	Graph(int vertex, int edge);
	Graph(Graph& g);

	void in(istream& stream);
	void out(ostream& stream) const;

	void addVertex();
	void addEdge(int a, int b);

	void delVertex(int index);
	void delEdge(int a, int b);

	vector<int> DFS(int start, int finish);

	vector<int> BFS(int start, int finish);

	void wayOut() const;
};

istream& operator>>(istream& stream, Graph& g);

ostream& operator<<(ostream& stream, Graph& g);

#endif
```

Содержание файла Graph.cpp

```
#include "Graph.h"

using namespace std;

// конструктор по-умолчанию
Graph::Graph() {
	vertexCount = 0;
	edgeCount = 0;
}

// конструктор с параметром
```

```
Graph::Graph(int vertex, int edge) {
	vertexCount = vertex;
	edgeCount = edge;
	matrix.resize(vertexCount);
	visited.resize(vertexCount);
}

// конструктор копий
Graph::Graph(Graph& g) {
	vertexCount = g.vertexCount;
	edgeCount = g.edgeCount;
	matrix = g.matrix;
	visited = g.visited;
}

// метод ввода объекта класса грав через поток
void Graph::in(istream& stream) {
	int a, b, i;
	matrix.clear();
	way.clear();
	visited.clear();
	if (&stream == &cin) {
		cout << "Введите количество вершин и ребер графа соответственно: ";
		stream >> vertexCount >> edgeCount;
		cout << endl << "Введите " << edgeCount << " пар вершин, которые
соеденены ребрами:" << endl;
		matrix.resize(vertexCount);
		visited.resize(vertexCount);
		for (i = 0; i < edgeCount; i++) {
			stream >> a >> b;
			matrix[a].push_back(b);
			matrix[b].push_back(a);
		}
	}

	else {
		stream >> vertexCount >> edgeCount;
		matrix.resize(vertexCount);
		visited.resize(vertexCount);
		for (i = 0; i < edgeCount; i++) {
			stream >> a >> b;
			matrix[a].push_back(b);
			matrix[b].push_back(a);
		}
	}
}

// метод вывода в консоль
void Graph::out(ostream& stream)const {
	int i, j;
	for (i = 0; i < vertexCount; i++) {
		stream << i << "| ";

		for (j = 0; j < matrix[i].size(); j++) {
			stream << matrix[i][j] << " ";
		}
		stream << endl;
	}

}

// добавить новую вершину в граф
void Graph::addVertex() {
	vertexCount++;
```

```
	matrix.resize(vertexCount);
	visited.resize(vertexCount);

	int edge, a;
	cout << "Введите количество ребер выходящих из этой новой вершины: ";
	cin >> edge;
	edgeCount += edge;

	if (edge != 0) {
		cout << "Введите " << edge << " вершин которые связаны с новой: ";
		for (int i = 0; i < edge; i++) {
			cin >> a;
			matrix[a].push_back(vertexCount - 1);
			matrix[vertexCount - 1].push_back(a);
		}
	}
}

// добавить ребро между двумя заданными вершинами графа
void Graph::addEdge(int a, int b) {
	matrix[a].push_back(b);
	matrix[b].push_back(a);
}

// удалить все связи заданной вершины графа
void Graph::delVertex(int index) {
	matrix[index].clear();
	for (int i = 0; i < vertexCount; i++) {
		for (auto j = matrix[i].begin(); j != matrix[i].end(); j++) {
			//cout << *j + 1 <<" ";
			if (*j == index) {
				matrix[i].erase(j);
				break;
			}
		}
	}
}

// удалить ребро между двумя заданными вершинами графа
void Graph::delEdge(int a, int b) {
	for (auto i = matrix[a].begin(); i != matrix[a].end(); i++) {
		if (*i == b) {
			matrix[a].erase(i);
			break;
		}
	}

	for (auto i = matrix[b].begin(); i != matrix[b].end(); i++) {
		if (*i == a) {
			matrix[b].erase(i);
			break;
		}
	}
}

// функция возвращающая путь между двумя вершинами графа
// путь находится по алгоритму поиска в глубину
vector<int> Graph::DFS(int start, int finish) {
	vector<int> res;
	visited[start] = 1;
	way.push_back(start);

	if (start == finish)
		return way;
```

```
	for (int to : matrix[start]) {
		if (!visited[to]) {
			res = DFS(to, finish);
			if (!res.empty())
				return res;
		}
	}

	way.pop_back();
	return {};
}


const int INF = 1e9;
// функция возвращающая путь между двумя вершинами графа
// путь находится по алгоритму поиска в ширину
vector<int> Graph::BFS(int start, int finish) {
	vector<int> dist(vertexCount, INF);
	vector<int> from(vertexCount, -1);
	queue<int> q;

	dist[start] = 0;
	q.push(start);

	int vertex;
	while (!q.empty()) {
		vertex = q.front();
		q.pop();

		for (int to : matrix[vertex]) {
			if (dist[to] > dist[vertex]) {
				dist[to] = dist[vertex] + 1;
				from[to] = vertex;
				q.push(to);
			}
		}
	}

	way.clear();
	for (int i = finish; i != -1; i = from[i]) {
		way.push_back(i);
	}

	reverse(way.begin(), way.end());

	return way;
}

// вывести путь в консоль
void Graph::wayOut() const {
	for (auto i = way.begin(); i != way.end(); i++) {
		cout << *i;
		if(i != way.end()-1)
			cout << " -> ";
	}
	cout << endl;
}

// перегрузка оператора ввода
istream& operator>>(istream& stream, Graph& g) {
	g.in(stream);
	return stream;
}
```

```
// перегрузка оператора вывода
ostream& operator<<(ostream& stream, Graph& g) {
	g.out(stream);
	return stream;
}
```

Содержание файла main.cpp

```
#include<iostream>
#include<Windows.h>
#include<fstream>
#include"GraphMatrix.h"
#include<conio.h>
#include<chrono>

#include"Graph.h"

using namespace std;
using namespace chrono;

int main(void) {
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	char key1, key2;
	bool flag1 = true, flag2 = true;
	int start, finish, index;

	// переменные необходимые, чтобы засекать время
	steady_clock::time_point begin, end;
	microseconds duration;

	ifstream file;

	Graph grph;

	// пользовательский интерфейс
	cout << "\t\t\t< Добро пожаловать! >" << endl;
	do {
		cout << endl << "---------------------------------->| Главное меню |<--
------------------------------" << endl << endl;
		cout << "1 режим - заполнить граф из файла и выбрать способ нахождения
пути;" << endl
			<< "2 режим - сформировать граф из данных пользователя и выбрать
способ нахождения пути." << endl
			<< "Любая другая клавиша для завершения работы программы." <<
endl;
		cout << endl << "----------------------------------------------------------
------------------------------" << endl;
		key1 = _getch();

		switch (key1) {
		// 1 режим - работа с графом из файла
		case '1':
			file.open("input.txt");
			file >> grph;

			cout << "Граф:" << endl << grph << endl;
			cout << "\tВыберите способ нахождения пути:"<< endl
				<< "\t1 - Алгоритм поиска в глубину;" << endl
				<< "\t2 - Алгоритм поиска в ширину;" << endl
				<< "\tЛюбая другая клавиша для выхода в главное меню." <<
endl;

			key2 = _getch();
```

```
                    if (key2 == '1' || key2 == '2') {
                          cout << "Введите стартовую вершину и конечную
соответственно: ";
                          cin >> start >> finish;
                    }

                    // нахождение пути выбраным способом
                    switch (key2) {
                    case '1':
                          cout << endl << "Путь найденый алгоритмом поиска в
глубину:" << endl;
                          begin = high_resolution_clock::now();
                          grph.DFS(start, finish);
                          end = high_resolution_clock::now();
                          duration = duration_cast<microseconds>(end - begin); //
расчет времени нахождения пути

                          grph.wayOut();
                          cout << "Время: " << duration.count() << " мкс.";
                          break;

                    case '2':
                          cout << endl << "Путь найденый алгоритмом поиска в
ширину:" << endl;
                          begin = high_resolution_clock::now();
                          grph.BFS(start, finish);
                          end = high_resolution_clock::now();
                          duration = duration_cast<microseconds>(end - begin);

                          grph.wayOut();
                          cout << "Время: " << duration.count() << " мкс.";
                          break;

                    default:
                          break;
                    }
                    file.close();
                    break;

                    // 2 режим - работа с пользовательским графом
              case '2':
                    cin >> grph;
                    flag2 = true;

                    do {
                          cout << "\tВыберите действие:" << endl
                                << "\t1 - Добавить вершину;" << endl
                                << "\t2 - Добавить ребро;" << endl
                                << "\t3 - Удалить вершину;" << endl
                                << "\t4 - Удалить ребро;" << endl
                                << "\t5 - Найти выход." << endl << endl;

                          key2 = _getch();

                          // редактирование графа
                          switch (key2) {
                          case '1':
                                grph.addVertex();
                                cout << "Вершина добавлена." << endl << endl;
                                break;

                          case '2':
```

```
                                cout << "Введите индексы двух вершин, между которыми
добавить ребро: ";
                                cin >> start >> finish;
                                grph.addEdge(start, finish);
                                cout << "Ребро добавлено." << endl << endl;
                                break;

                        case '3':
                                cout << "Введите индекс вершины, связи которой
хотите удалить: ";
                                cin >> index;
                                grph.delVertex(index);
                                cout << "Все связи вершины удалены." << endl <<
endl;
                                break;

                        case '4':
                                cout << "Введите индексы двух вершин, между которыми
удалить ребро: ";
                                cin >> start >> finish;
                                grph.delEdge(start, finish);
                                cout << "Ребро удалено." << endl << endl;
                                break;

                        case '5':
                                flag2 = false;
                                break;

                        default:
                                break;
                        }

                } while (flag2);

                cout << "Граф:" << endl << grph << endl;
                cout << "\tВыберите способ нахождения пути:" << endl
                        << "\t1 - Алгоритм поиска в глубину;" << endl
                        << "\t2 - Алгоритм поиска в ширину;" << endl
                        << "\tЛюбая другая клавиша для выхода в главное меню." <<
endl << endl;

                key2 = _getch();

                if (key2 == '1' || key2 == '2') {
                        cout << "Введите стартовую вершину и конечную
соответственно: ";
                        cin >> start >> finish;
                }

                // нахождение пути выбранным способом
                switch (key2) {
                case '1':
                        cout << endl << "Путь найденый алгоритмом поиска в
глубину:" << endl;
                        begin = high_resolution_clock::now();

                        grph.DFS(start, finish);
                        end = high_resolution_clock::now();
                        duration = duration_cast<microseconds>(end - begin);

                        grph.wayOut();
                        cout << "Время: " << duration.count() << " мкс.";
                        break;
```

```
                    case '2':
                          cout << endl << "Путь найденый алгоритмом поиска в
ширину:" << endl;
                          begin = high_resolution_clock::now();
                          grph.BFS(start, finish);
                          end = high_resolution_clock::now();
                          duration = duration_cast<microseconds>(end - begin);

                          grph.wayOut();
                          cout << "Время: " << duration.count() << " мкс.";
                          break;

                    default:
                          break;
                    }
                    break;

              default:
                    flag1 = false;
                    break;
              }
        } while (flag1);
}
```

# ПРИЛОЖЕНИЕ В

# ОТЧЕТ АНТИПЛАГИАТА

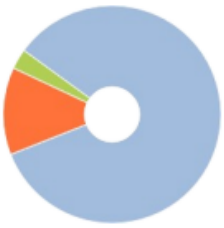
АНТИПЛАГИАТ
ОБНАРУЖЕНИЕ ЗАИМСТВОВАНИЙ

Отчет предоставлен сервисом «Антиплагиат» - http://sfedu.antiplagiat.ru

## Отчет о проверке

**Автор:** Решетнева М Ф
**Проверяющий:** Подгайко Алексей Андреевич
**Название документа:** 98580
**Организация:** Южный федеральный университет

### РЕЗУЛЬТАТЫ ПРОВЕРКИ



Совпадения: 13,1%
Оригинальность: 83,78%
Цитирования: 3,12%
Самоцитирования: 0%

«Совпадения», «Цитирования», «Самоцитирования», «Оригинальность» являются отдельными показателями, отображаются в процентах и в сумме дают 100%, что соответствует проверенному тексту документа.

Есть подозрения на следующие группы маскировки заимствований: Сгенерированный текст на страницах: 4, 5, 6, 19, 20, 21, 22, 23, 24, 25

- **Совпадения** — фрагменты проверяемого текста, полностью или частично сходные с найденными источниками, за исключением фрагментов, которые система отнесла к цитированию или самоцитированию. Показатель «Совпадения» – это доля фрагментов проверяемого текста, отнесенных к совпадениям, в общем объеме текста.
- **Самоцитирования** — фрагменты проверяемого текста, совпадающие или почти совпадающие с фрагментом текста источника, автором или соавтором которого является автор проверяемого документа. Показатель «Самоцитирования» – это доля фрагментов текста, отнесенных к самоцитированию, в общем объеме текста.
- **Цитирования** — фрагменты проверяемого текста, которые не являются авторскими, но которые система отнесла к корректно оформленным. К цитированиям относятся также шаблонные фразы; библиография; фрагменты текста, найденные модулем поиска «СПС Гарант: нормативно-правовая документация». Показатель «Цитирования» – это доля фрагментов проверяемого текста, отнесенных к цитированию, в общем объеме текста.
- **Текстовое пересечение** — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.
- **Источник** — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.
- **Оригинальный текст** — фрагменты проверяемого текста, не обнаруженные ни в одном источнике и не отмеченные ни одним из модулей поиска. Показатель «Оригинальность» – это доля фрагментов проверяемого текста, отнесенных к оригинальному тексту, в общем объеме текста.

Обращаем Ваше внимание, что система находит текстовые совпадения проверяемого документа с проиндексированными в системе источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности совпадений или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

### ИНФОРМАЦИЯ О ДОКУМЕНТЕ

**Номер документа:** 240
**Тип документа:** Не указано
**Дата проверки:** 14.12.2024 13:17:11
**Дата корректировки:** Нет
**Количество страниц:** 34
**Символов в тексте:** 39296
**Слов в тексте:** 5088
**Число предложений:** 2332